Performance Management Using a Rapid and Practical Method

Lamia Djoudi and William Jalby

Université de Versailles Saint-Quentin, France,
Tel: +33(0)1 39 25 43 43 - Fax +33 (0)1 39 25 40 57
Contact: lamia.djoudi@prism.uvsq.fr

Abstract

Nowadays, compilers contain a large number of optimizations. These optimizations are based on a set of heuristics that are not guaranteed to be effective to improve the performance or the code size. Several techniques have been proposed to optimize either performance or code size. To find a trade-off between these two issues is a difficult matter and thus may be expensive. Program performance is tightly linked to the assembly code, this is even emphasized on EPIC architectures. Assessing precisely quality of compiled code is essential to deliver high performance.

In this paper, we propose an approach which allows us to find a trade-off between code quality, code size and performance. This trade-off gives us different possibilities to (1) choose the best transformations in order to improve performance and/or code quality, (2) improve the code quality in order to deliver high performance, (3) improve performance without the code explosion, and (4) give performance estimates .

With this approach, if we choose a transformation, our system provides the performance estimates , loop size and code size. If we propose a code size, our system gives us the corresponding transformation that we must apply and also the performance estimates.

Key words: Iterative compilation, Code optimization, Performance, Tools, Unrolling factors, Loops

1. Introduction

Nowadays, compilers contain a large number of optimizations. These optimizations, often come from the research world, go through various stages of refinement and applicability, before being included in real world compilers. These optimizations are based on a set of heuristics that are not guaranteed to be effective to improve performance metrics. Moreover, these heuristics do not allow us to improve the performance and at the same time avoid the code size explosion problem.

Designing these heuristics is generally difficult. The heuristics must be specific to each implementation of the instruction set architecture. They are also dependent on changes made to the compiler. Several techniques have been proposed to optimize either performance or code size. To find a trade-off between these two issues is a difficult matter and thus may be expensive.

In this paper, we propose an approach which allows us to find a trade-off between code quality and performance. This trade-off gives us different possibilities to (1) choose the best transformations in order to improve

performance and/or code quality, (2) improve the code quality in order to deliver high performance, (3) improve performance without the code explosion, and (4) give performance estimates .

With this approach, if we choose a transformation, our system provides the performance estimates , loop size and code size. If we propose a code size, our system gives us the corresponding transformation that we must apply and also the performance estimates.

Our approach is based on iterative compilation. Iterative compilation is an approach that relies on the generation of many different versions of the same code to find out the best among them. Loops generated by iterative compilation are therefore good candidates. Iterative compilation is decomposed usually into two steps: (1) Generate multiple optimized versions of the same code. (2) For each loop, we compare the performance of its different versions (either by a model or by a dynamic evaluation) and build a program combining them. In our research, we focus to give out the advantages of the two steps. Based on first step, we generate different best versions for each hot loop. These versions do not explode the code size and must improve code quality and performance. Based on second step, our approach:

- enables one execution to (1) execute the best loop versions in the same program and at the same time, (2) select for each interval of iterations the best version, and (3) find a trade-off between code size, code quality and the performance.
- proposes different optimizations (on high level or low level) to have a trade-off between code size, code quality and performance estimates.

Iterative compilation strongly depends on the quality of the information acquired from the execution stage. Current profiling mainly reports behavioral properties of the executed code, such as cycle counts and at best hardware counters. By doing profiling, it tends to neglect code structures and to flatten complex hierarchies (control flow graphs, call graphs, data dependency graphs).

The critical problem of iterative compilation remains the containment of the search space combinatory explosion. With traditional iterative compilation all the optimization decisions are taken at a very high level, either in a pre-processing tool or, at best, within the compiler front-end. An originality of our method is to deport part of optimizations from the driver to a post-compiler evaluation stage. The idea is that a close examination of the code after the compilation is the best location to decide if some transformations should be applied or not.

The static/dynamic modules of MAQAO[1] tool analyze plain assembly code and detect if some optimization fails, or if due to some obscure compiler decision, the resulting code contains under-performing patterns.

This static/dynamic analysis is the basic information that *MAQAOAdvisor* uses to achieve a trade-off between code quality, code size and performance. By combining static and dynamic analysis, we centralize all low level performance and build correlations.

Assessing precisely quality of compiled code is essential to deliver high performance. Nowadays this issue is mostly tackled by using hardware counters and dynamic profiling. Static analysis, as we aim to illustrate in this paper, can achieve similar results at a much lower cost and with a better accuracy.

By analyzing the assembly code, our approach gives the first decision about the code quality and which transformation should be applied to improve the quality of assembly code and by consequence improve its performance. MAQAOAdvisor, a key MAQAO(Modular Assembly Quality Optimizer) module drives the optimization process through assembly code analysis and performance evaluation. It implements a set of rules to help end-user to detect and understand performance problems. It performs comprehensive profiling, hot-loop and hot-spot detection, fast evaluation and guides local optimizations.

1.1 Motivating Example

```
DO 1 I=2,N
  CP1=1./(CP(I)-CI(I)*CS(I-
  CS(I)=CS(I)*CP1
  CSM(I)=(CSM(I)-CI(I)*CSM(
1 CONTINUE
```

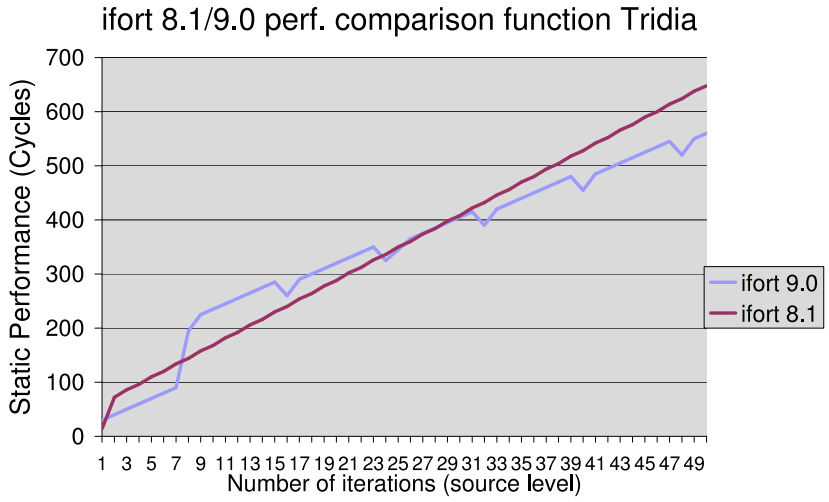


Figure 1. Tridia code and its performance with ICC version 8.1 and 9.0. Moving from version 8.1 to 9.0, ICC has changed part of its code generation policy. None of the two different versions is optimal over the whole possible range of iterations.

Loop optimization is a critical part in the compiler optimization chain. A routinely stated rule is that 90% time of the execution time is spent in 10% part of the code. Another rule, implicitely used by the community, is that the number of iterations for loops in scientific code is *large*. Consequently, loops are often unrolled, pipelined deeply and data streams aggressively prefetched.

However, optimizations for asymptotic behavior involve a part of risk. For instance in software pipeline, depth is always increased if it can reduce the Initiation Interval. This yields to codes which deliver poor performance when the number of iterations is limited. Figure 1 clearly illustrates the trade-off that the compiler has to handle on a simple vector loop named Tridia. ICC 8.1, first unrolls this loops two times and generates a software pipeline of depth 2. While ICC 9.0 unrolls this loops 8 times, then applies software pipeline. The corresponding tail code is also software pipelined.
The corresponding performance evaluation is:

- ICC 9.0: $65 \times \frac{N}{8} + 130$ (unrolled 8 times) and $10 \times (N \bmod 8) + 20$ for tail code.
- ICC 8.1: $24 \times \frac{N}{2} + 48$ (unrolled 2 times) and $14 \times (N \bmod 2)$ for tail code.

As illustrated by figure 1, ICC 9.0 choice is justified for asymptotic performance but is doubtful when the number of iterations is small.

The rest of this paper is organized as follows: Section 2 details the extracting the information from the original code. Section 3 details the guided optimization. Section 4 presents case studies. Section 5 presents related work. And we conclude in Section 6.

2. Extracting the Information from the Original Code

Gathering data and statistics is necessary for a performance tool, but it remains only a preliminary stage. The most important step is to build a comprehensive summary for end-user and extract manageable information. To sum-up this section, MAQAO computes the structure of the assembly code and enables the application of specific analysis and expert knowledge to be integrated inside the tool. Code pattern detection can be combined with hotspot detection or other dynamic analysis. As a result, it has a great potential in revealing the possible flaws of the code generation. MAQAO identifies the optimizations done (or not) by the compiler, its expert system (*MAQAOAdvisor*) drives the optimization processes through assembly code analysis and performance evaluation. *MAQAOAdvisor* suggests the optimizations, improves the code quality and the performance. It implements a set of rules to help end-user to detect and understand performance problems. MAQAO modules are centered around a core module and a database which is ultimately the place where every piece of information is stored. The database is in charge of ensuring data persistence and also offers a standardized storage format.

2.1 Dynamic Information

MAQAOPROFILE[3] allows us to give a precise weight to all executed loops, therefore underscoring hotspots. Correlating this information provides the relevant metrics:

(i) Identifying the hotpath at run-time which passes through the whole program where the application spends the most of its time is a key for understanding application behavior.

(ii) Monitoring trip count is very rewarding, by default most of compiler optimizations target asymptotic performance. Knowing that a loop is subjected to a limited number of iterations allow us to choose the optimizations which is characterized by a cost function.

A loop is a candidate for optimization if it is a counted loop e.g, Fortran DO loops, or FOR loops in C.

2.2 Static Information

Close inspection of assembly code is a real mine of information. An important work is to filter out essential pieces from low interest part of the code.

MAQAO can coalesce the instructions per family (e.g. integer arithmetics, load instructions) and count them on a per basic block basis.

For each loop, MAQAO can give a summary of instructions that have determined as being of special interest like spill/fill instructions. It can also detect if there are functions calls or register pressure.

MAQAO computes several metrics which should trigger attention. An example of static analyses included in MAQAO:

- *Issues cost per iteration*, basically it reports only the number of cycles needed to transmit all loop instructions. This value corresponds to the number of bit stop in the loop body. Jointly with cycle costs, this metric allows us to evaluate the cost of data dependences for the loop. A large gap induced by data dependency hints that the loop should be unrolled more aggressively or targeted by other techniques to increase the available parallelism.
- *Cycle cost per iteration*, is a reference point to estimate the effectiveness of dynamic performance. This value corresponds to the number of static cycles corresponding to loop body predicted by compiler.
- *Theoretical cycle bound per iteration*[10], knowing whether a loop is compute or memory bound is a powerful indicator of the kind of optimization techniques to use. Typically compute bound loops imply that lots of cycles are available to tolerate memory latency problem.

From this last computation, MAQAO produces an important ratio $R1$. This first ratio evaluates the matching between static bounds and observed performance.

2.3 Combining Static and Dynamic Information

By combining static and dynamic analysis, MAQAOAdvisor detects the value undecidable by a pure static scheme and gives more information to take the best decision. For example:

1. The versioning option of MAQAO gives a versioning summary for each hot loop(See figure 2). The idea is to perform a study of different versions based on the number of iterations, to decide which is the best version for each interval of iterations. With this study, we classify the versions as function of the number of iterations, and choose for each interval of iterations the best one in order to improve parallelism in the original code or in the new optimized code (very interesting to improve the compositional versioning[11, 12]).
2. To find trade-off between quality and performance it is interesting to calculate the second ratio (R2). A possible model for the dynamic cost function is: $c_2(N) =$ number of cycles executed
   $R2$ is the ratio between static cycles and dynamic cycles. This second ratio answers the question: Does the static code represent a good dynamic behaviour ?

To take a decision to what we do, *MAQAOAdvisor* combines the information like $R1$ and $R2$ values of one or more versions for each hot loop. For example, *MAQAOAdvisor* guides user to use hardware counters when there is a large difference between the ratios $R1$ and $R2$. In this case, executing a simple script in MAQAO, *MAQAOAdvisor* combines the hardware counters and MAQAO results for guiding user to take a decision. For example to solve the cache misses, *MAQAOAdvisor* can propose one of the decisions:
The memory reuse, by modifying the stride of the loop or aggregating the data.
optimization of the cache, by taking a copy of data or a blocking cache in order to decrease TLB misses.
Recovery of Data access latencies, by adding a pragma in source code or modify the prefetch distance in assembly code. This modification is still in progress in the compositional approach implemented in MAQAO.

3. Guided Optimization

With traditional iterative compilation all the optimization decisions are taken at a very high level, either in a pre-processing tool or, at best, within the compiler front-end. The idea is that a close examination of the code after the compilation is the best location to decide if some transformations should be applied or not. *MAQAOAdvisor* helps end-user to navigate through her code and isolate the particularly important or suspicious pieces of code.

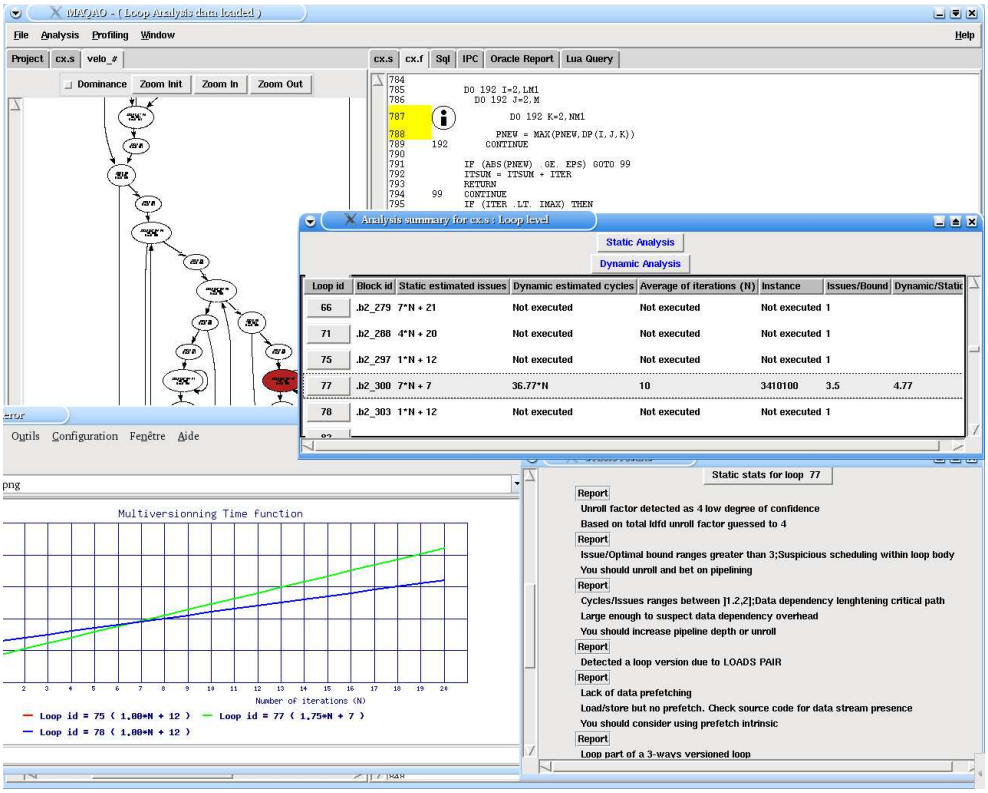


Figure 2. Cx Becnh: Loop versions of loop 787 (source line). Compiler generates three assembly versions (loop id: 75, 77 78) but it executes just the loop version 77. With MAQAO versioning option, we remark that this loop is better just for the first iteration. The best solution that it must combine the two versions (77 and 78 (or 75)). It executes the loop 77 for the first for the first seventh iterations and loop 78 (or 75) for the rest of iterations.

For these isolated pieces which are the hot inner loops, *MAQAOAdvisor* provides as many guideline as possible to help the decision making process. This ”guided-profile” allows us to understand the compiler optimizations and guides us to improve code quality and performance.

3.1 Code Quality

For a single source loop, compiler produces one or several assembly versions in the same assembly code. MAQAO modules perform a study for each version. The static module of MAQAO gives the $R1$ value and the code and loop sizes. Based on $R1$ value, *MAQAOAdvisor* proposes two main solutions:

1- If an assembly version has $R1 \leq 1.2$ and there is no problem of spill /fill, no *check* instructions and no function calls , *MAQAOAdvisor* informs user that this version has a good quality and:

- guides her to stopped the optimization process if there is just one assembly version.
- It saves the code size, the loop size and the static cycles for each version in order to compare between the good versions. This comparison may be beneficial while applying the ordering algorithm of the best versions.

2- If $R1 \geq 1.3$, *MAQAOAdvisor* generates the first guided optimization. It combines the static and dynamic analysis of the original version of each hot loop. At this level, *MAQAOAdvisor* takes a first deduction of compiler optimizations and proposes to apply different transformations for several hot loops in assembly or in source level in order to improve code quality and by consequence it's performance.

For example, it can propose inserting of one or a combination of "pragma" in order to avoid (1) the register pressure in order to avoid the spill/fill, (2) the *check* instructions, these instructions mean that the compilers had taken a bad decision, (3) the functions calls can decrease performance.

When *MAQAOAdvisor* orients user to generate different versions of each hot loop, the generation of several versions of the original code is automatic. At this level, MAQAO has the possibility to perform a global study (static analysis, profiling, ...) for all versions at the same time. This automatic process is the "mode project" in MAQAO and the user can have a comparison and a guiding report to choose the best transformation for each hot loop.

To generate more versions, we focus on loop unrolling, software pipelining and prefetch distance.

For unrolling, our approach is based on iterative compilation to determine simultaneously optimal code size and the best unrolling factor to improve performance. It proposes a model to determine the maximum value of unrolling factor (UFmax). It proposes also a linear function that studies the original code generated by the compiler and the code size generated after applying the best unrolling factor. Since we do not know what will the compiler generate, we propose to apply different values of unrolling factors which are less or equal to the maximum value of unrolling factor. The original motivations of including loop unrolling because it is a fundamental technique for generating the long instruction sequences required by VLIW machines[29]. In this paper, four factors are investigated: the compiler transformations, unrolling factor, the code size and the hot loops.

For software pipelining, combining unrolling and software pipelining may produce a loop with an initial interval closer to the lower bound, thus a faster loop.

For prefetch, data prefetching cuts by a large amount the read/write latency of memory accesses. Tuning the prefetch distance is highly dependent on the total number of iterations. For small loops, the prefetch distance is too large for the prefetching to be effective. In this case, removing the prefetches may free resources for a better ILP, therefore increasing performance. An estimation of the prefetch distance can be made on the basis of three things: (1) the number of cycles per iteration, (2) the stride of the data access (usually equal to the number of bytes consumed per iteration) and (3) latency from main memory. For the data to be delivered prior to its

consumption the prefetch distance:

$$Prefetch\ distance > latency \times \frac{\frac{stride}{128}}{cycles\ per\ iteration}$$

At the code source level, the code size is not changed because we add one (or two) line only to apply pragma unrolling (or pragma unrolling and software pipelining) before each inner loop. At the assembly code level, our method will not have to deal with the code size explosion problem.

3.1.1 Selecting the maximum value of the unrolling factor under constraint of code size

Let *CS* be the code size of the original assembly code. Let us suppose that if we have one version of hot loop in the code, its size is *LS*. Let α_0 be the ratio LS and CS, that $\alpha_0 = \frac{LS}{CS}$

Our goal is to avoid the code size explosion problem after unrolling, that means, we must choose the smallest value of (*Unrolling Factor times Loop Size*).

At the source code, unrolling *UF* times consists of replacing the loop body by a larger body composed of *UF* copies of the original. At the low level, if the compiler applies unrolling, it tries to apply parallelism, and the size of unrolling loop is: $LSU \leq UF * LS$

$CSU \geq (CS - LS) + LSU$

$CSU \geq (CS - \alpha_0 * LS) + UF * LS$

$\frac{CSU}{CS} \geq 1 + \alpha_0(UF - 1)$

We suppose $\beta = \frac{CSU}{CS}$, then

$$UF_max \leq \frac{\beta - 1 + \alpha_0}{\alpha_0} \tag{1}$$

3.1.2 Selecting the Unrolling Factors

Let *t(CS)* be the execution time corresponding to code size of the original assembly code. Let us suppose that if we have one version of hot loop in the code, its execution time is *t(LS)*

γ_0 is the ratio t(LS) and t(CS): $\gamma_0 = \frac{t(LS)}{t(CS)}$

Our goal is to avoid the code size explosion after unrolling, that means, we must choose the smallest $Unrolling\ Factor\ *\ Loop\ Size$.

The execution time of unrolling code depends on the execution time of the unrolling loop.

$t(CSU) = t(CS) - t(LS) + t(LSU) = 1 - \gamma_0 + \frac{t(LSU)}{t(LS)}$

At the source code, unrolling UF times consists of replacing the loop body by a larger body composed of UF

copies of the original. At the low level, if the compiler applies unrolling, it tries to apply parallelism, and the execution time of unrolling loop is: $t(LSU) \leq UF * t(LS)$

We suppose : $\Gamma = \frac{t(CSU)}{t(CS)}$*, then*

$$UF \geq \Gamma + \gamma_0 - 1 \tag{2}$$

3.1.3 Improving Code Quality

In order to direct the search in the optimization space, a GLPK[2], a free easy solver is used, that we integrate in MAQAO. GLPK takes into account the criteria proposed by our approach. Our constraints are presented by linear functions. We presented our constraints in the following form:

Let i: 1...n the number of hot loops.

Let j: 1...n the number of transformations to be applied. In this paper, our transformations are software pipelining or unrolling. For unrolling, we can generate for each hot loop different versions corresponding to unrolling factor less than the UFmax

SC_ij*: is the static cycles of loop i after applying transformation j*

LS_ij*: is the loop size of loop i after applying transformation j*

x_ij*: is a binary variable:*

$$(x_{ij}) \quad = \quad \begin{cases} 1 : if\ we\ apply\ transformation \\ 0 : else \end{cases}$$

$R1_ij$*, the first ratio (issues/bound) of loop i after applying transformation j*

If we fix the loop sizes, our system is presented as:

$$(S1) \quad : \quad \begin{cases} \sum_{i=1}^{n} \sum_{j=1}^{m} x_ij \geq 1 \\ \sum_{i=1}^{n} \sum_{j=1}^{m} LS_ij.x_ij \leq Cste_1 \\ \sum_{i=1}^{n} \sum_{j=1}^{m} R1_ij.x_ij \leq Cste_2 \\ min \sum_{i=1}^{n} \sum_{j=1}^{m} SC_ij.x_ij \end{cases}$$

And if we fix the unrolling factor and the static cycles, the solver must find the the minimum function of the loop size:

$$(S2) \quad : \quad \begin{cases} \sum_{i=1}^{n} \sum_{j=1}^{m} x_i j \geq 1 \\ \sum_{i=1}^{n} \sum_{j=1}^{m} SC_i j.x_i j \leq Cste_1 \\ \sum_{i=1}^{n} \sum_{j=1}^{m} R1_i j.x_i j \leq Cste_2 \\ min \sum_{i=1}^{n} \sum_{j=1}^{m} LS_i j.x_i j \end{cases}$$

$Cste_1$: is the static cycle that can be fixed by user. To fix this value, user can use the enequation (2) in order to avoid the code size explosion or choose an impossible static cycles.

$Cste_2$: is the code size that can be fixed by user. To fix this value, user can use the enequation (1) in order code size explosion or choose an impossible static cycles.

MAQAOAdvisor also gives an estimation to the user that can be used it.

*The advantage of our system is that it is a fast one, which means that if we have n*m variables we will have n+3 constraints. In assembly code, the number of hot loops is not very big and the number of optimizations that we propose is not very big also. Our method is easy to use because:*

- The detection of static and dynamic information is automatic.

- Adding pragmas to generate new versions is automatic.

- Finding trade-off between code quality and performance is also automatic.

3.2 Managing Performance-Code Size

Our approach is based on iterative compilation for determining simultaneously optimal code size and the best transformations to improve performance of the compiler without the code size explosion.

For each loop, we can have the number of iterations, the execution time or cpu cycles, the optimizations applied by compiler (software pipelining, unrolling, ...), the unrolling factor applied by compiler for each assembly version, which assembly version has been executed, and the loop size.

For the code, we have a report including the size of assembly and source codes, the execution time, the hot loops, the hotpath, and the number and type of instructions, etc.

Based on all these information, MAQAOAdvisor proposes the best transformation for each loop.

3.2.1 Classification of the Best Versions

The analysis of all generating versions to decide which is the best or what kind of transformation user must take to have the best performance is taken at the second level of the MAQAOAdvisor. It is possible that the compiler

may not improve the code quality, so MAQAOAdvisor orients user to the second decisions.

All these versions are estimated according to a cost model for a given iteration range. The idea is to perform a general study and to propose the best version for each interval of number of iterations depending on the number of iterations (N). The advantage of this classification is demonstrated in[11, 12].

3.2.2 Trade-off Performance-Code Size

To find this trade-off, our method can generate two graphs:

1. $UFmax(\beta)$*: if the user chooses an interval of iterations or one value, she can find* $UFmax$ *and then she can find* Γ *and we can provide the estimate* $\frac{t(CSU}{t(CS)}$
2. $UF(\Gamma)$*: if she should find the execution time (CPU cycles), she must choose an unrolling factor value and automatically she gets the* β

For each case, we have the $UF \leq UFmax$ *with the execution time and the code size.*

4. CASE STUDIES

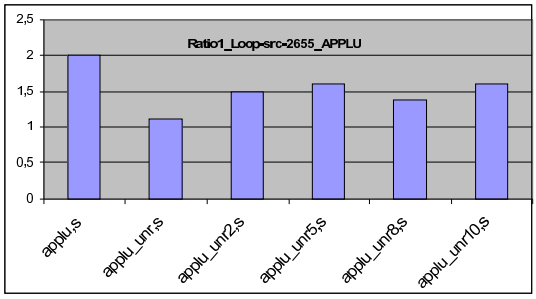
In this section, we evaluate our proposed technique. We consider JACOBI code, and two benchmarks from the SPECFP2000.

Experiments were run on a BULL Itanium 2 Novascale system, 1.6GHz, 3MB of L3 on the software side, codes were compiled using Intel ICC/IFORT 9.1.

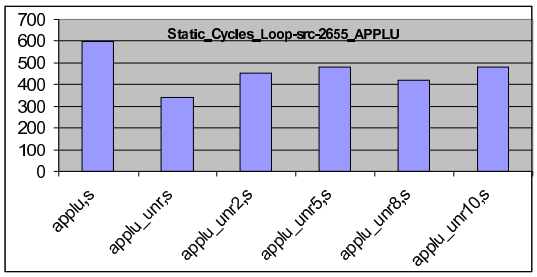
173.APPLU: *The first example presents the results of our approach is 173.APPLU. It is a benchmark from SPECFP2000 which leads to the performance evaluation of the solver for five coupled parabolic/elliptic partial differential equations.*

The first level of our approach demonstrates that:

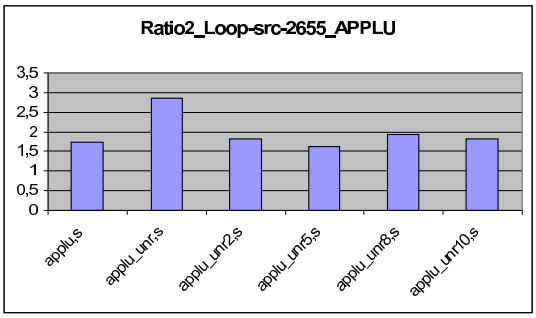
- *173.applu contains one important hot loop and their static infomations (R1, number of registers used, ...) show that it n important loop. 290 is the trip count.*
- $R1$ *= 2 and the MAQAOAdvisor guided, proposes to unroll this loop.*
- *Solved the equation 2 in section 3 for this loop, we have* $UFmax = 10$*.*
- *We choose the unrolling factors: 2,5,8 and 10. And also use the pragma "pragma unroll" where compiler unroll this loop four times.*



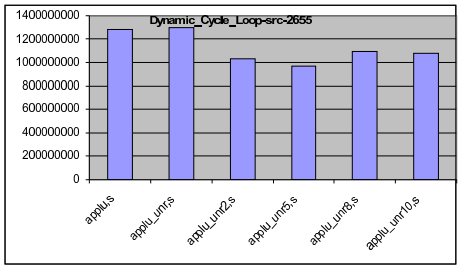
(a) The first Ratio using selective iteration compilation



(b) Static cycles of the selected versions



(c) Ratio2 corresponding to the hot loop in applu code.



(d) Dynamic cycles of the selected versions

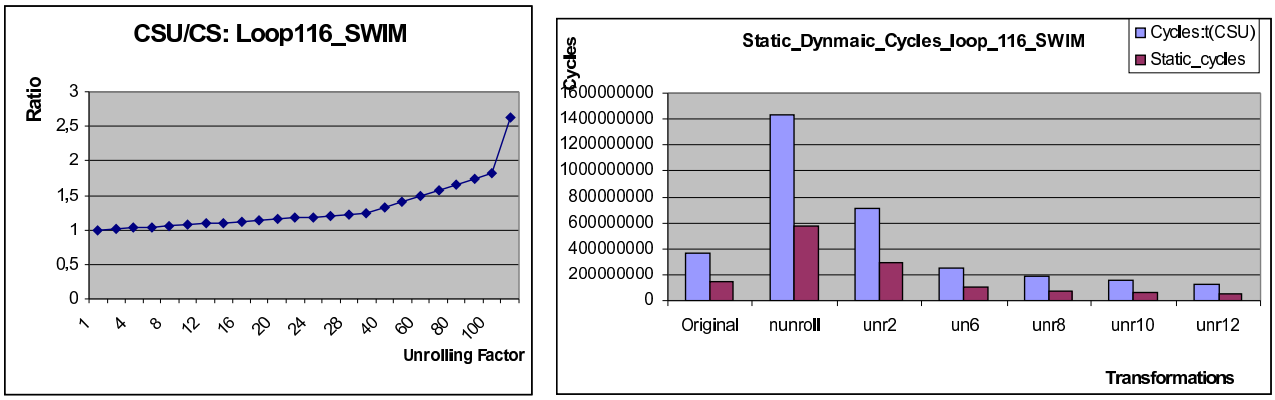
Figure 3. Trade-off performance-code size to select the best version

- *In the following study, we automatically generate some versions of the code corresponding of the unrolling pragmas. As depicted in figure 3 (a) and in figure 3 (b) , the first ratio of the original code is the bad ratio. The version (applu_unr corresponding to pragma unroll. The compiler unroll four times the loop. At this level, we can decide that the version corresponding of unrolling 4 time is the best. To verify if this version is the best one, we have instrumented these versions.*
- *The second ratio (figure 3 (c)) and dynamic cycles (figure 3 (d)) demonstrate that the second ratio of version unroll 10 is the best but the dynamic cycles of the version unroll 5 is the best.*

Introduce all these informations to the GLPK solver, it finds a trade-off and decide the version unroll 5 is the best one.

171.SWIM: *To demonstrate how user can choose the unrolling factor depending on the code size and vice-versa, we have test the 171.swim SPEC benchmark. Specially we have used the loop 116 (source line) because compiler basically has used an unrolling* $(UF = 4)$*. For example, if user selects some unrolling factor (user is free to choose any value), MAQAOAdvisor gives a summary of all transformations. For example based on code size (respectively the loop size) before and after unrolling, we have the graph (see figure 4 (a)).*

Based on this graph user can choose the maximum value of the unrolling factor. Like for the above example,



(a) Ratio original code size and unrolling code size

(b) Static and dynamic Cycles

Figure 4. Selecting a version based on unrolling factor, static and dynamic cycles.

MAQAOAdvisor proposes the different value less than UFmax. For loop 116, we have the unrolling factor, 2, 6, 8, 10. 4 is the unrolling factor corresponding to the original loop. In this level, user can choose the mode project of MAQAO to have all dynamic and static information of these different versions.

User can follow the same process of the above example or generate a comparison of static and dynamic cycles to choose the best version. See figure 4 (b)).

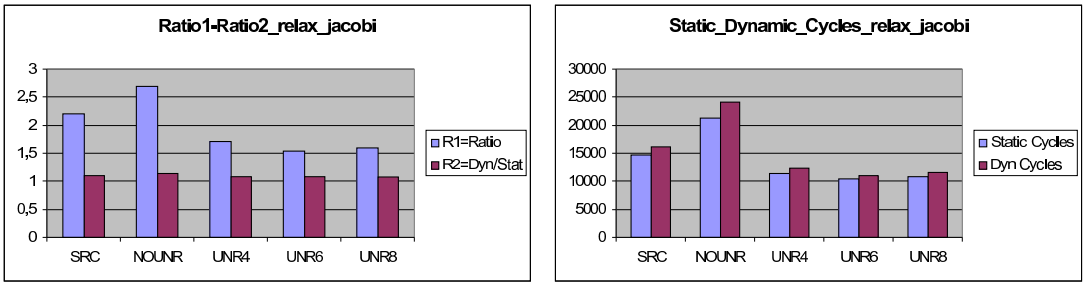
JACOBI: *Jacobi code solves the Helmholtz equation on a regular mesh, using an iterative Jacobi method with over-relaxation. The first level of our approach demonstrates that jacobi contains one important hot loop (loop source line 2655). This level allows us to generate some versions of this loop using pragma. Introducing all guided-profile important information to the GLPK[2] solver, it finds a trade-off and decides the version unroll 6 is the best one (see figure 5 (a) and (b)).*

To improve the compositional versioning technique[11, 12], we used the MAQAO versioning option described in section 2.3. It is an extra stage to select automatically and quickly the best version for each interval of iterations. Applying these two technique to JACOBI code, we have improved the performance (see Figure 5 (c))

5. *Related Work*

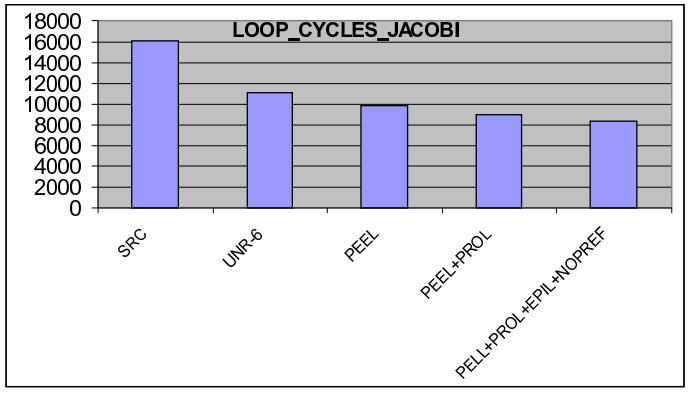
There is a lot of reserch on automatically selecting for the best compiler optimization. the work [4, 5, 6] is based on iteratevely enabling certain optimization, running the compiled program and, based on its performance, deciding on a new optimization setting. Compilers apply a complete fixed pipeline of optimizations from the source code to the binary[13]. Parello et al.[7], Cavazos et al.[8], use hardware counters to generate the heuristics in order to predict good optimizations.

Our work, concentrates upon post-compiler, hence we are sure that the compiler does not undo optimizations.



(a) Static guided-profile

(b) Dynamic guided-profile



(c) Compositional versioning of jacobi

Figure 5. Hot file (relax_jacobi), hot loop (2655 source line): (a) the first and the ratios of the selected versions of the hot loop. (b) the static cycles and the dynamic cycles of the selected versions. (c) CPU cycles for different compositional versioning: (a) loop 23 (source line) in relax_jacobi.

For our approach, we prove that static analyis is an important step to propose a good optimization. Hardware counters are used to complete the MAQAO process. They are implemented in MAQAO. MAQAOAdvisor guides user to use hardware counters when there is a large difference between the ratios $R1$ and $R2$.

The impact prefetch can be deduced by MAQAO analysis. If [9] proposes to insert prefetch or changing prefetching distance dynamically depending on hardware counters informations, we propose the same study based on static and dynamic analysis of MAQAO to improve the performance[11].

There is a lot of reserach[4, 16] that proves the benefit of iterative compilation. The purpose of iterative compilation is to converge toward an optimal combinations of transformations in an exhaustive[15, 14] or handled way. A large research effort was focused on decreasing the search space[6, 17]. For example, machine learning[18, 19] aims at speeding up iterative process by correlating program features with corresponding performance. During an initial step, the model learns from a set of training programs. Therefore, this method is very sensitive to initial benchmarks chosen as training program. Model-driven optimizations (e.g. cache models [21, 22]) allow to cut the search space complexity by preselecting high potential transformations and their related parameters. Model approach tends to be too restricted. For instance, Fraguela et al. [20, 24] use a model

only to solve locality problem.

Among the interesting works on iterative compilation, a major finding brought by extensive search [25] is that performance is widely sensitive to minor changes in the transformation sequence. Exploring the local neighborhood of a fixed transformation sequence can lead to near optimal performance.

In this paper, we propose to add an extra phase of the process of iterative compilation in order to reduce the space of iterative compilation. For one execution of source code, MAQAOAdvisor guides user to generate a small number of versions for each hot loop. This number is limited by the maximum value of unrolling factor, the code size and the performance.

For versionning, [26, 27, 28] use simple version selection mechanisms according to the input run-time function or loop parameters. Our vesionning is benificial because we choose for each interval of iterations, the best version. In the final code, we are sure that we have only the best versions.

In this paper, to generate different versions, we use unrolling, software pipeling and prefetching. Unrolling has the advantage that it can be applied to any loop, and can be done profitably both at the high and low levels.

The benefits of unrolling have been studied on several different architectures. Unrolling is a fundamental technique for generating the long instruction sequences required by VLIW machines [29].

In addition to its use in compilers, many software libraries for matrix computation contain loops that have been hand-unrolled for improved performance [30].

Whaley and Dongara [31] have described a system for generation highly optimized versions of a set of Basic Linear Algebra Subroutines (BLAS). This system can probe the underlying hardware to find optimal values for blocking factors, unrolling factors etc.

Based on unrolling and software pipelining, it has been observed that loop unrolling can also have a negative effect on a program's performance when it is not used judiciously. And since compilers are based on heuristics, good unrolling factor may not be properly applied. Applying excessive unrolling can lead to run-time performance degradation due to extra registers spills/fills. It can also overflow a small first-level instruction cache. Stephenson et al.[33], described a machine learning to help compiler to classify the optimal unrolling factor. His technique is usefull to the problem of heuristic tuning. He used a multi-class classification to improve compiler decisions and to determine optimal unrolling factor. Monsifrot et al.[14] consider binary classification and leaving the choice of unroll factor up to a compiler heuristics. Our method has the same goal but we use a post compilation that collect all compiler information and decide which is the best unrolling factor.

Our method will not exceed the number of available registers and respects the cache locality.

The problem of automatically selecting unroll factors for nested loop has been addressed in past work. [32, 34] their results for loop kernels are impressive, and make a convincing case for leaving the task of selecting unroll factors to the compiler rather than the programmer. However, their results for full applications are less convincing. Their objective is also to balance floating-point and memory-access instructions.

The main objective of [35] is to reduce the execution time. When they find the best unrolling factor, they unrolled loop on high-level. For our approach, we fix the maximum value based on code quality and performance.

Sakar[35] has presented how unrolling can be performed when he unrolled the outer loop R times and unroll the nested loop only one time. The first problem is that he had the remainder loop. He can authorize the number of registers spill if there is registers spill in original code . The number of unrolling factors for each loop is not limited. With his objective function, he can reduce the execution time. For us, we are interested in innermost and hot loops. With our approach, we avoid the spill/fill, the register pressure. For an input data, our system tries to choose the versions without remainder loop.

6. Conclusion

In this paper, we have proposed a method to decrease the time and space of iterative compilation, to find a trade-off between performance and code size. Based on assembly level, profiling information, our system is able to select the best optimizations among a list defined by the user or using directives supported by the compiler.

One of the keys of our approach is the ordering algorithm that gives the user the classification of the best transformation proposed by MAQAOAdvisor and based on her critiria. It gives the possibility to the user to choose the transformations (respectively the code size), and our system provides the code size and performance estimates (respectively the best transformations and estimation performance).

One of the extensions of this algorithm is to apply it to give the possiblity to choose a combination of transformations. While writing this paper, this algoritm is applied to choose a combination of "pragma".

The advantage of our technique is fully automatic and all processes are in the same system. MAQAO modules (static analysis, profiling,...) are complementary. One of the main future work is the generation of models of kernels and saving them as mathematical components and as DDG (For DDG models, the work is still in progress). The idea is to have a database to save the models of kernels in order to propose best optimizations and best performance quickly for future applications.

The goal is to improve MAQAOAdvisor to a real expert system. This system contains a knowledge base

containing accumulated experience and a set of rules for applying the knowledge base to each particular kernel. It can be enhanced with additions to the knowledge base or to the set of rules. From an implementation point of view, our work is still in progress.

References

[1] L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J.-T. Acquaviva and W. Jalby MAQAO: Modular Assembler Quality Analyzer and Optimizer for Itanium 2 In Workshop on EPIC architectures and compiler technology, *San Jose, 2005.*

[2] http://www.gnu.org/software/glpk

[3] L. Djoudi, D. Barthou, O. Tomaz, A. Charif-Rubial, JT. Acquaviva and W Jalby The Design and Architecture of MAQAOPROFILE: an Instrumentation MAQAO Module Workshop on architectures and compiler technology, *San Jose, 2007.*

[4] M. Haneda, P. M. W. Knijnenburg, Harry A. G. Wijshoff Automatic Selection of Compiler Options Using Non-parametric Inferential Statistics. In IEEE PACT 2005: 123-132

[5] Z.Pan, R. Eigenmann Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning In Proceedings of the International Symposium on Code Generation and Optimization table of contents, Pages: 319 - 332, 2006

[6] S. Triantafyllis and M. Vachharajani and N. Vachharajani and D. August Compiler optimizationspace exploration In Proceedings of the 03 International Symposium on Code Generation and Optimization, pp. 204–215, March 2003

[7] D. Parello, O. Temam, A. Cohen, J.-M. Verdun Toward a Systematic, Pragmatic and Architecture-Aware Program Optimization Process for Complex Processors In Supercomputing, IEEE, Nov 2004

[8] J.Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. O'Boyle and O. Temam Rapidly Selecting Good Compiler Optimizations using Performance Counters In Proceedings of the 5th Annual International Symposium on Code Generation and Optimization (CGO), San Jose, USA, March 2007

[9] R. H. Saavedra and D. Park Improving the Effectiveness of Software Prefetching with Adaptive Execution In PACT 1996

[10] L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J.-T. Acquaviva and W. Jalby Exploring Application Performance: a New Tool for a Static/Dynamic Approach. In Los Alamos Computer Science Institute Symp., *Santa Fe, NM, 2005.*

[11] L. Djoudi, J.-T. Acquaviva, D. Barthou Compositional Approach applied to Loop Specialization In Euro-Par Conference, volume 4641 of Lect. Notes in Computer Science, pages 268-279, Rennes, Aug. 2007. Springer-Verlag

[12] Submitted paper

[13] R. Allen and K. Kennedy Optimizing Compilers for Modern Architectures In Morgan and Kaufman, 2002

[14] A. Monsifrot and F. Bodin and R. Quiniou A Machine Learning Approach to Automatic Production of Compiler Heuristics In Artificial Intelligence: Methodology, Systems, Applications, pages 41–50, 2002

[15] K. D. Cooper and D. Subramanian and L. Torczon Adaptive optimizing compilers for the 21st century In J. of Supercomputing, 2002

[16] T. Kisuki and P. Knijnenburg and M. O'Boyle and H. Wijshoff Iterative compilation in program optimization In Proc. CPC'10 (Compilers for Parallel Computers) 35–44, 2000

[17] G.G. Fursin and M.F.P. O'Boyle and P.M.W. Knijnenburg Evaluating Iterative Compilation",book In Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computers (LCPC'02)", pages=305-315, year=2002

[18] Grigori Fursin and Albert Cohen and Michael O'Boyle and Oliver Temam Quick and practical run-time evaluation of multiple program optimizations In journal = Transactions on High-Performance Embedded Architectures and Compilers, "2006", volume = "1", number = "1", pages = "13-31"

[19] F. Agakov and E. Bonilla and J.Cavazos and B.Franke and G. Fursin and M.F.P. O'Boyle and J. Thomson and M. Toussaint and C.K.I. Williams Using Machine Learning to Focus Iterative Optimization In Proceedings of the 4th Annual International Symposium on Code Generation and Optimization (CGO), 2006

[20] B. Fraguela and R. Doallo and J. no and E. Zapata, A compiler tool to predict memory hierarchy performance of

scientific codes In Parallel Computing. citeseer.ist.psu.edu/fraguela04compiler.html, 2004

[21] P. Knijnenburg and T. Kisuki and K. Gallivan and M. O'Boyle The Effect of Cache Models on Iterative Compilation for Combined Tiling and Unrolling In Proc. FDDO-3, pages 31-40, 2000

[22] T. Kisuki and P. Knijnenburg and M. O'Boyle Incorporating cache models in iterative compilation for combined tiling and unrolling In Technical Report no. 2000-10, LIACS, Leiden University, 2000.

[23] P. Carribault and A. Cohen and W. Jalby Deep Jam: Conversion of Coarse Grain Parallelism to Intruction-Level and Vector Parallelism for Irregular Applications In The Fourteenth International Conference on Parallel Architectures and Compilation Techniques (PACT'05), IEEE Computer Society

[24] Basilio B. Fraguela and Ramon Doallo and Emilio L. Zapata Automatic Analytical Modeling for the Estimation of Cache Misses In PACT '99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques, IEEE Computer Society

[25] K. D. Cooper and T. Waterman Investigating Adaptive Compilation using the MIPSPro Compiler In Proc. of the Symp. of the Los Alamos Computer Science Institute, October, 2003

[26] M. Byler, M. Wolfe, J. R. B. Davies, C.r Huson, B. Leasure Multiple Version Loops In ICPP 1987, pages: 312-318

[27] K. Cooper and M. W. Hall and K. Kennedy Procedure Cloning In Proceedings of the 1992 IEEE International Conference on Computer Language, 1992

[28] Pedro C. Diniz and Martin C. Rinard Dynamic feedback: an effective technique for adaptive computing In In Proc. PLDI, pages 71–84; 1997.

[29] J. R. Ellis Bulldog: A Compiler for VLIW Architectures. MIT Press, pp. 180-184(1986).

[30] Jack Dongarra, A. R. Hinds Unrolling Loops in FORTRAN. Softw., Pract. Exper. 9(3): 219-226 (1979)

[31] R. Whaley and J. Dongarra Automatically Tuned Linear Algebra Software Technical Report UT CS-97-366, LAPACK Working Note No.131, University of Tennessee, 1997

[32] Steve Carr and Yiping Guan Unroll-and-jam using uniformly generated sets nternational Symposium on Microarchitecture 1997

[33] M. Stephenson, S. Amarasinghe Predicting Unroll Factors Using Supervised Classification In Proceedings of International Symposium on Code Generation and Optimization. San Jose, California. March 2005

[34] Steve Carr and Ken Kennedy Improving the Ratio of Memory Operations to Floating-Point Operations in Loops ACM Transactions on Programming Languages and Systems, 1994

[35] Vivek Sakar Optimized Unrolling of Nested Loops International Journal of Parallel Programming, 2004